Coded Matrix Chain Multiplication

Xiaodi Fan*, Angel Saldivia†, Pedro Soto*, Jun Li‡
*Graduate Center, City University of New York
†School of Computing and Information Sciences, Florida International University
‡Queens College and Graduate Center, City University of New York

***Abstract*—The matrix multiplication is a fundamental building block in many machine learning models. As the input matrices may be too large to be multiplied on a single server, it is common to split input matrices into multiple submatrices and execute the multiplications on different servers. However, in a distributed infrastructure it is common to observe stragglers whose performance is lower than other servers at some time. In order to mitigate the adversarial effects of potential stragglers, various coding schemes for the distributed matrix multiplication have been recently proposed. While most existing works have only considered the simplest case where only two matrices are multiplied, we investigate a more general case in this paper where multiple matrices are multiplied, and propose a coding scheme that the result can be directly decoded in one round, instead of in multiple rounds of computation. Compared to completing the matrix chain multiplication in multiple rounds, our coding scheme can achieve significant savings of completion time by up to 90.3%.**

***Index Terms*—matrix chain multiplication, entangled polynomial code, distributed computing, linear regression**

I. Introduction

The matrix multiplication is a fundamental operation for solving various learning-based problems. With the ever growing sizes of learning models and datasets, the sizes of the matrix multiplication in the models are also increasing. It has become challenging to execute the matrix multiplication on a single server when input matrices are from large datasets. Therefore, it is common to split the job of matrix multiplication to multiple tasks which can be executed on different servers in parallel.

However, it is well known that servers in a distributed infrastructure, *e.g.*, in a cloud, can exhibit faulty behaviors [1] due to load imbalance, resource contention, or hardware issues, *etc*. For example, it has been observed that virtual machines on Amazon EC2 may be $5\times$ slower than others of the same type [2], [3]. Moreover, up to more than 100 server failures can happen on a daily basis in a cluster of Facebook with 3000 servers [4], [5]. Therefore, if some tasks are running on such servers, *i.e.*, stragglers, they will become the bottleneck of the job. Even one single straggler can significantly slow down the overall progress of the whole job, as the completion of the whole job depends on the completion of all of its tasks.

A naive method that mitigates the adversarial effects of stragglers is to replicate each task on multiple servers, so that the job can be completed as long as one of them runs on a non-straggling server. However, it incurs an excessive amount of resource consumption. To tolerate any r stragglers, all tasks need to be replicated on $r+1$ servers. On the other hand, coding-based methods have been proposed where the result of the job can be decoded from a certain number of *coded* tasks [2], [3], [6], [7]. As illustrated in Fig. 1a, in order to calculate AX, we first split A into two submatrices A_0 and A_1 so that $AX = \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} X = \begin{bmatrix} A_0X \\ A_1X \end{bmatrix}$. Then one additional coded task can be created as $(A_0+A_1)X$, such that any two of the three tasks can recover the result of AX. Compared to replication which needs two additional tasks to tolerate one single straggler, we save the number of workers by 25% in Fig. 1b. Hence, the coded matrix multiplication enjoys a higher level of straggler tolerance with much fewer additional tasks.

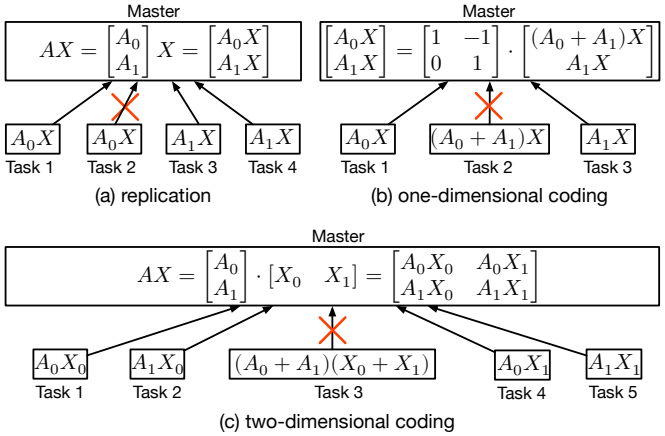

Fig. 1. Examples of the coded matrix multiplication.

In order to reduce the size of the task, existing works for the coded matrix multiplication have been evolved from one-dimensional coding [3], [6], [8], [9] where only one matrix is split as shown in Fig. 1b, to two-dimensional coding [10], [11], [12], [13], [14] where both two matrices can be split, one vertically and the other horizontally, as illustrated in Fig. 1c. Comparing the tasks in Fig. 1b and those in Fig. 1c, we can see that the size of the tasks in Fig. 1c is further reduced by 50%. On the other hand, we need to have five tasks to tolerate one straggler in Fig. 1c, as the overall result needs to be decoded from any four tasks.

More generally, the two input matrices can be split both horizontally and vertically into $p_0 \times p_1$ and $p_1 \times p_2$ submatrices. Fig. 1c, for example, corresponds to a special case of $p_0 = p_2 = 2$ and $p_1 = 1$. A coded task will then multiply two

coded matrices that are linear combinations of submatrices in the two input matrices. Therefore, we can see that the input matrices can be divided into more and more submatrices of smaller sizes, and then each task can be computed with less time. To the best of our knowledge, the overall result of the matrix multiplication can only be decoded with the results of at least $p_0p_1p_2 + p_1 - 1$ tasks [7], [15].

However, existing works on the coded matrix multiplication have been focusing on the multiplication of only two matrices, while in practice there are various learning-based algorithms requiring the result of the matrix chain multiplication, *i.e.*, multiplying multiple matrices together. For example, the matrix chain multiplication is required in the randomized singular value decomposition [16], which is a core in the principal component analysis and are applied widely in various machine learning models [17], [18].

With existing coding techniques above, the matrix chain multiplication can only be completed by multiple rounds of matrix multiplications. At least one input matrix in each round must be based on the result of the previous round. Although the sequence of multiplication can be determined using dynamic programming to minimize the overall computational complexity [19], the result of each round still needs to be decoded and encoded again for the next round, making the job completion time increase linearly with the number of rounds.

In this paper, we propose a general coding framework for the matrix chain multiplication where the job can be finished with just one round of tasks. Assume that there are m matrices M_i, $i = 0, \ldots, m-1$, and we aim to calculate their multiplication $\prod_{i=0}^{m-1} M_i$. Although Dutta *et al.* [15] have also discussed coding for the matrix chain multiplication, the input matrices must be partitioned with specific patterns. Our coding scheme, instead, supports to split the matrix in a more general way, where M_i can be split into any p_i partitions vertically and p_{i+1} partitions horizontally, and hence M_i will be divided into p_ip_{i+1} submatrices, *i.e.*,

$$M_i = \begin{bmatrix} M_i^{0,0} & \cdots & M_i^{0,p_{i+1}-1} \\ \vdots & \ddots & \vdots \\ M_i^{p_i-1,0} & \cdots & M_i^{p_i-1,p_{i+1}-1} \end{bmatrix}.$$

Each task will then be a chain multiplication of m coded matrices encoded from the submatrices in $M_0, \ldots, M_{m-1}$.[1] We prove that with our coding scheme, the overall result of $\prod_{i=0}^{m-1} M_i$ can be decoded with the results of any $\prod_{i=0}^{m+1} p_i + \prod_{i=1}^{m} p_i - 1$ tasks. In particular, we will see that the coding scheme proposed in [7] can be considered as a special case of $m = 2$.

We implement and evaluate our coding scheme using OpenMPI running on Microsoft Azure and find that the job completion time of the matrix chain multiplication can be improved by up to 90.3%, compared to multiplying them one by one in multiple rounds. We further run linear regression implemented based on the coded matrix chain multiplication, and demonstrate that it can save completion time by up to 51.7%, compared to the same job with entangled polynomial codes.

[1]Since a coded task still calculates the matrix chain multiplication, dynamic programming can also be applied on each task to minimize its complexity, regardless of the coding scheme. Therefore, we focus on the coding scheme only in this paper, instead of the order of multiplication in each task.

II. Related Work

Stragglers are a common issue in distributed systems which can be incurred by various reasons such as network congestion, source contention, load imbalance, and hardware failures [1], [20], [21]. Conventionally, such stragglers can be mitigated by periodically writing checkpoints of intermediate results into a reliable storage device [22], [23]. When a straggler is detected [24], [25], the affected tasks can be relaunched on a replacement server with previous results loaded from a recent checkpoint. However, it incurs a significant amount of time and network traffic to write checkpoints. As the network bandwidth is already a source of bottleneck in a distributed infrastructure, writing checkpoints periodically may further increase the job completion time.

On the other hand, we can add redundant tasks in the job in advance to tolerate potential stragglers. The naive way of adding redundant tasks is replicating each task on multiple servers [14], [24], [26], [27], [28], [29]. Compared with replication, adding coded tasks have been shown to tolerate the same number of stragglers with fewer tasks. Given a coding scheme, we define its recovery threshold as the smallest number of arbitrary tasks that can be decoded to obtain the overall result. The recovery threshold should be as low as possible to tolerate the same number of stragglers with fewer tasks.

Lee *et al.* [3], for the first time, proposed a coding scheme for matrix-vector multiplication, where only the matrix is split vertically and encoded. Such one-dimensional coding has also been constructed with sparse coding [6] or rateless coding [30]. However, when there are two large matrices to multiply, the coded tasks may still be too large to run on a single server.

Two-dimensional coding can split both the two input matrices and encode them into coded tasks. There are two kinds of designs for two-dimensional coding. One is based on product code where a task encoded by a one-dimensional code can be further divided and encoded into more coded tasks [10], [12], [13], [31]. However, such coding schemes require that the group of tasks encoded from the same one-dimensional coding task must be decodable, *i.e.*, the number of non-straggling servers must reach the recovery threshold of the one-dimensional code, making its recovery threshold far from the optimum, as all stragglers may fall into the same group. On the other hand, polynomial codes are the only two-dimensional coding that achieves the optimal recovery threshold [14].

Two-dimensional coding can only divide each input matrix in one dimension only, either vertically or horizontally. Three-dimensional coding, including PolyDot codes [15] and entangled polynomial codes [7], can divide the two input matrices both vertically and horizontally. When the two input matrices

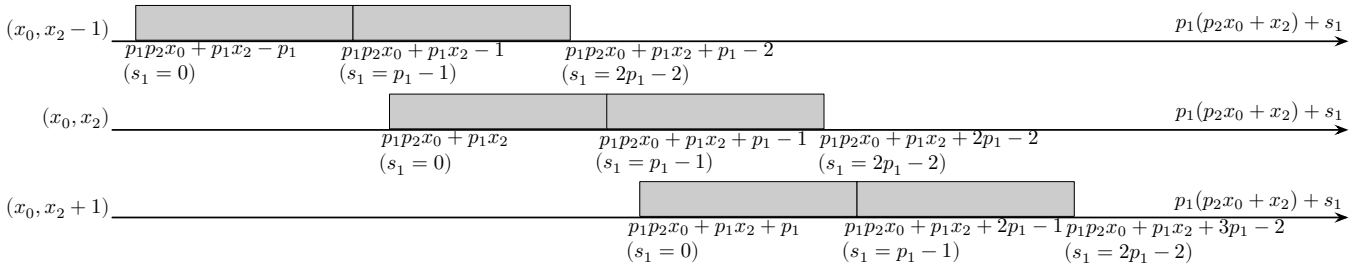

Fig. 2. Entangled exponents of X in $f_0(X)f_1(X)$.

are split into $p_0 \times p_1$ and $p_1 \times p_2$ submatrices, PolyDot codes can only support a special case of $p_0 = p_1$ while entangled polynomial codes can support any feasible values of p_0, p_1, and p_2.

In this paper, we consider a more general operation, *i.e.*, the distributed matrix chain multiplication, Although Dutta *et al.* have extended the code construction into the matrix chain multiplication [15], their code can only be constructed when the partitions of input matrices follow specific patterns, *i.e.*, the odd matrices and even matrices among the m input matrices must be partitioned in the same way, respectively. The code constructed in this paper, however, can support any arbitrary and valid partitions in all m input matrices. The recovery threshold in [15] can also be seen as a special case of our general recovery threshold.

III. Background: Entangled Polynomial Code ($m = 2$)

Before demonstrating our coding scheme for the matrix chain multiplication, we first give a brief review of entangled polynomial codes [7], a special case of our coding scheme with $m = 2$. We will construct our code for the matrix chain multiplication based on entangled polynomial codes. In this case, the multiplication of M_0 and M_1 can be written as

$$M_0M_1 = \begin{bmatrix} \sum_{x_1=0}^{p_1-1} M_0^{0,x_1}M_1^{x_1,0} & \cdots & \sum_{x_1=0}^{p_1-1} M_0^{0,x_1}M_1^{x_1,p_2-1} \\ \vdots & \ddots & \vdots \\ \sum_{x_1=0}^{p_1-1} M_0^{p_0-1,x_1}M_1^{x_1,0} & \cdots & \sum_{x_1=0}^{p_1-1} M_0^{p_0-1,x_1}M_1^{x_1,p_2-1} \end{bmatrix},$$

where we can see that there are p_0p_2 submatrices. With an entangled polynomial code, coded tasks are constructed to obtain such p_0p_2 submatrices. Each server runs a task that calculates $f_0(X)f_1(X)$, where the value of X is different from that in any other tasks. In particular,

$$f_0(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_1=0}^{p_1-1} M_0^{x_0,x_1}X^{p_1p_2x_0+x_1},$$

and

$$f_1(X) = \sum_{x_2=0}^{p_2-1}\sum_{x_1=0}^{p_1-1} M_1^{p_1-1-x_1,x_2}X^{p_1x_2+x_1}.$$

Therefore, we have

$$f_0(X)f_1(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_2=0}^{p_2-1}\sum_{s_1=0}^{2p_1-2}\left(\sum_{x_1=\max\{0,s_1-p_1+1\}}^{\min\{p_1-1,s_1\}} M_0^{x_0,x_1}M_1^{p_1-1-s_1+x_1,x_2}\right)X^{p_1(p_2x_0+x_2)+s_1}. \quad (1)$$

From (1), we can see that $f_0(X)f_1(X)$ is a polynomial function of X of degree $p_0p_1p_2 + p_1 - 2$. Therefore, we can decode the coefficients of $f_0(X)f_1(X)$ with $p_0p_1p_2 + p_1 - 1$ such tasks, by a polynomial interpolation algorithm or Gaussian elimination, In particular, in (1), the coefficients of $X^{p_1(p_2x_0+x_2)+s_1}$ with $s_1 = p_1-1$ are $\sum_{x_1=0}^{p_1-1} M_0^{x_0,x_1}M_1^{x_1,x_2}$. Therefore, we can obtain the p_0p_2 desired submatrices in M_0M_1 after decoding.

It is interesting to note that when $s_1 \neq p_1 - 1$, the corresponding coefficients are *noise* coefficients, *i.e.*, they are not needed after decoding. As shown in Fig. 2, we cover the exponents of the terms in $f_0(X)f_1(X)$ with some specific values of x_0 and x_1. We can see that such exponents range between $p_1p_2p_0 + p_1x_2$ and $p_1p_2x_0 + p_1x_2 + 2p_1 - 2$. In particular, the term with the exponent $p_1p_2x_0+p_1x_2+p_1-1$, which corresponds to $s_1 = p_1 - 1$, has its coefficient as a desired submatrix $\sum_{x_1=0}^{p_1-1} M_0^{x_0,x_1}M_1^{x_1,x_2}$.

In $f_0(X)f_1(X)$, the exponents of noise coefficients will not interfere with the other desired coefficients with $s_1 = p_1 - 1$, although the values of s_1 can range between 0 and $2p_1 - 2$. If we compare the terms of (x_0, x_2) with those of $(x_0, x_2 - 1)$, we can see that the exponents of $(x_0, x_2 - 1)$ are all smaller than the exponent that corresponds to the desired submatrix, if $x_2 > 0$. On the other hand, the exponents of $(x_0, x_2 + 1)$ are all larger than that of the desired submatrix if $x_2 < p_2 - 1$. Hence, no other terms in $f_0(X)f_1(X)$ will have the same exponent as $p_1p_2x_0 + p_1x_2 + p_1 - 1$. We can also get the same result if $x_2 = 0$ or $x_2 = p_2 - 1$, *i.e.*, when $(x_0, p_2 - 1)$ goes to $(x_0 + 1, 0)$. Therefore, all desired submatrices with all possible values of (x_0, x_2) will also have their exponents, making sure that their values can be correctly obtained after decoding.

Moreover, the exponents of noise coefficients can overlap so that the degree of the polynomial can be reduced. In Fig. 2, we can see that except the exponent corresponding to $s_1 = p_1-1$, all the other exponents around can be matched with the same exponent above with (x_0, x_2-1) or lower with (x_0, x_2+1). As

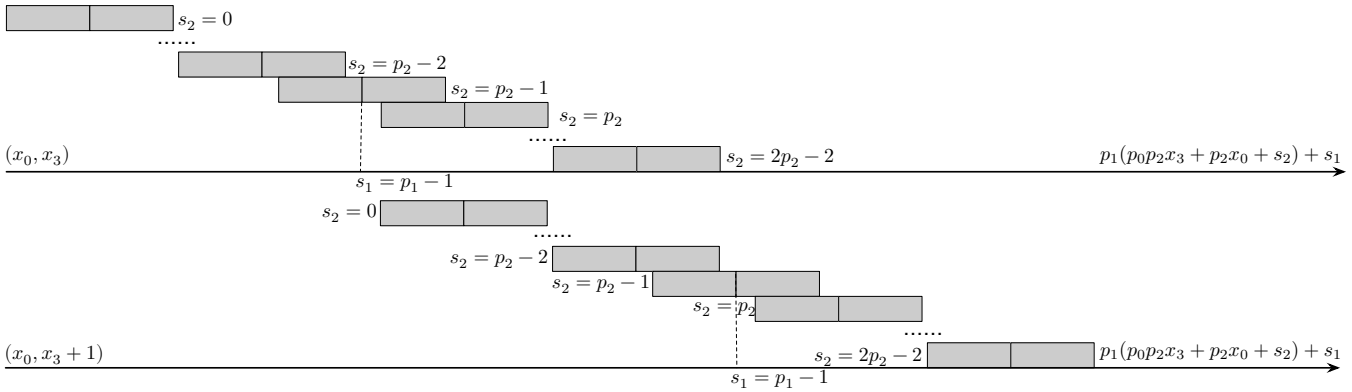

Fig. 3. Entangled exponents of X in $f_0(X)f_1(X)f_2(X)$.

their corresponding coefficients are noise, they can be added together without affecting the overall result after decoding. Therefore, compared to making all terms with different values of (x_0, x_2) have different exponents, entangled polynomial codes save the overall degree of $f_0(X)f_1(X)$ which also helps to achieve a low recovery threshold. In this paper, we will further utilize this property in order to achieve a low recovery threshold in the coded matrix chain multiplication.

IV. Coded Chain Multiplication of Three Matrices ($m = 3$)

We now start to construct our coding scheme for the matrix chain multiplication. For simplicity, we first present the construction for a special case of multiplying three matrices, *i.e.*, $m = 3$. We will present the code construction for a general value of m in Sec. V.

We show that the code for the multiplication of three matrices can be extended from the entangled polynomial code. Considering the case of $M_0M_1M_2$, a coded task will then be constructed as $f_0(X)f_1(X)f_2(X)$. Here, $f_0(X)$ and $f_1(X)$ remain the same as constructed with the corresponding entangled polynomial code constructed for M_0M_1, and we will now present how to construct $f_2(X)$.

Similar to the case of $m = 2$ in Sec. III, there will be p_0p_3 submatrices in $M_0M_1M_2$. In particular, if we define $\hat{M}_2 = M_0M_1$, we can divide $\hat{M}_2$ into p_0p_2 submatrices and have

$$M_0M_1M_2 = \hat{M}_2M_2 = \begin{bmatrix} \sum_{x_2=0}^{p_2-1} \hat{M}_2^{0,x_2}M_2^{x_2,0} & \cdots & \sum_{x_2=0}^{p_2-1} \hat{M}_2^{0,x_2}M_2^{x_2,p_3-1} \\ \vdots & \ddots & \vdots \\ \sum_{x_2=0}^{p_1-1} \hat{M}_2^{p_0-1,x_2}M_2^{x_2,0} & \cdots & \sum_{x_2=0}^{p_2-1} \hat{M}_2^{p_0-1,x_2}M_2^{x_2,p_3-1} \end{bmatrix},$$

where $\hat{M}_2^{x_0,x_2} = \sum_{x_1=0}^{p_1-1} M_0^{x_0,x_1}M_1^{x_1,x_2}$, $0 \le x_0 \le p_0 - 1$ and $0 \le x_2 \le p_2 - 1$.

We will now discuss how to construct $f_2(X)$ to obtain such p_0p_3 submatrices. Considering (1), we can rewrite $f_0(X)f_1(X)$ as:

$$f_0(X)f_1(X) \tag{2}$$
$$= \sum_{x_0=0}^{p_0-1}\sum_{x_2=0}^{p_2-1}\sum_{s_1=0}^{2p_1-2} C_2(x_0, x_2, s_1)X^{p_1(p_2x_0+x_2)+s_1} \tag{3}$$
$$= \sum_{s_1=0}^{2p_1-2}\left(\sum_{x_0=0}^{p_0-1}\sum_{x_2=0}^{p_2-1} C_2(x_0, x_2, s_1)X^{p_1(p_2x_0+x_2)+s_1}\right), \tag{4}$$

where $C_2(x_0, x_2, s_1) \triangleq \sum_{x_1=\max\{0,s_1-p_1+1\}}^{\min\{p_1-1,s_1\}} M_0^{x_0,x_1} M_1^{p_1-1-s_1+x_1,x_2}$. From Sec. III, we know that we are interested in the value of $C_2(x_0, x_2, s_1)$ if $s_1 = p_1 - 1$. Therefore, the submatrices in $\hat{M}_2$ have been encoded in $f_0(X)f_1(X)$ as

$$\hat{f}_2(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_2=0}^{p_2-1} C_2(x_0, x_2, p_1 - 1)X^{p_1(p_2x_0+x_2)+p_1-1} = \sum_{x_0=0}^{p_0-1}\sum_{x_2=0}^{p_2-1} \hat{M}_2^{x_0,x_2}X^{p_1(p_2x_0+x_2)+p_1-1}.$$

Now we reapply entangled polynomial codes to $\hat{M}_2M_2$, and encode M_2 as $f_2(X) = \sum_{x_3=0}^{p_3-1}\sum_{x_2=0}^{p_2-1} M_2^{p_2-1-x_2,x_3}X^{p_0p_1p_2x_3+p_1x_2}$. We will then get

$$\hat{f}_2(X)f_2(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_3=0}^{p_3-1}\sum_{s_2=0}^{2p_2-2}\left(\sum_{x_2=\max\{0,s_2-p_2+1\}}^{\min\{p_2-1,s_2\}} \hat{M}_2^{x_0,x_2}M_2^{p_2-1-s_2+x_2,x_3}\right)X^{p_1(p_0p_2x_3+p_2x_0+s_2)+(p_1-1)}.$$

Similarly, we are interested in the coefficients where $s_2 = p_2 - 1$. Therefore, the desired coefficients in

$f_0(X)f_1(X)f_2(X)$ are those with $s_1 = p_1-1$ and $s_2 = p_2-1$:

$$f_0(X)f_1(X)f_2(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_3=0}^{p_3-1}\sum_{s_2=0}^{2p_2-2}\sum_{s_1=0}^{2p_1-2} \left(\sum_{x_2=\max\{0,s_2-p_2+1\}}^{\min\{p_2-1,s_2\}} C_2(x_0,x_2,s_1)M_3^{p_2-1-s_2+x_2,x_3}\right) X^{p_1(p_0p_2x_3+p_2x_0+s_2)+s_1}. \quad (5)$$

We show the entangled exponents of $f_0(X)f_1(X)f_2(X)$ in Fig. 3. Given a fixed (x_0, x_3), we can see that the exponents of the corresponding terms are entangled in the same way as entangled polynomial codes. In this case, however, the exponents are also further entangled with the exponents with (x_0, x_3+1). Similar to entangled polynomial codes, although $p_1s_2+s_1$ can vary from 0 to $2p_1p_2-2$, and the exponents of X in (5) will increase by p_1p_2 when (x_3, x_0) goes to (x_3, x_0+1), we can still see that the desired coefficients in the middle will not interfere with noise coefficients around, which overlap with other noise coefficients. As shown in Fig. 3, given x_0 and x_3, the only desired coefficient has $s_1 = p_1 - 1$ and $s_2 = p_2 - 1$, while all other noise coefficients can overlap with each other with different values of s_2. When we change x_3 to $x_3 + 1$, the exponents of coefficients with $s_2 \neq p_2 - 1$ will further be entangled with those of previous coefficients of (x_0, x_3), while the desired coefficient still enjoys its unique exponent.

Given the entangled exponents above, we can see that the degree of $f_0(X)f_1(X)f_3(X)$ is $p_0p_1p_2p_3 + p_1p_2 - 2$, as the exponents of X range from 0 to $p_0p_1p_2(p_3 - 1) + p_1p_2(p_0 - 1)+p_1(2p_2-2)+(2p_1-2) = p_0p_1p_2p_3+p_2p_2-2$. Therefore, the recovery threshold is $p_0p_1p_2p_3+p_1p_2-1$, and the desired submatrices can be found in p_0p_3 of its coefficients with $s_1 = p_1 - 1$ and $s_2 = p_2 - 1$.

V. General Coded Matrix Chain Multiplication

A. Code construction

We now generalize the code construction for the matrix chain multiplication with any m matrices, $m \geq 2$. We define an encoding function Ω_m that generates $f_0(X), \ldots, f_{m-1}(X)$, *i.e.*, $(f_0(X), \ldots, f_{m-1}(X)) = \Omega_m(M_0, \ldots, M_{m-1})$.

Following the method in Sec. IV, the general encoding function Ω_m can be constructed recursively in Alg. 1. We define $P_a^b = \prod_{i=a}^b p_i$. In particular, if $a > b$, we define $P_a^b = 1$.

Note that such $f_0(X), \ldots, f_{m-1}(X)$ can be constructed before encoding and then the encoding process will be directly evaluating the value of such m polynomials with a unique value of X. Given $(f_0(X), \ldots, f_{m-1}(X)) = \Omega_m(M_0, \ldots, M_{m-1})$, we will obtain a coded task $F_m(X) = \prod_{i=0}^{m-1} f_i(X)$. Moreover, the coefficients in $F_m(X)$ can also be decoded by interpolation or Gaussian elimination, as in the entangled polynomial code.

We will now analyze the properties of such a coding scheme as follows, including its recovery threshold and correctness.

Algorithm 1 Construction of $\Omega_m(M_0, \ldots, M_{m-1})$

1: **if** $m = 2$ **then**
2: $f_0(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_1=0}^{p_1-1} M_1^{x_0,x_1}X^{p_1p_2x_0+x_1}$
3: $f_1(X) = \sum_{x_2=0}^{p_2-1}\sum_{x_1=0}^{p_1-1} M_2^{p_1-1-x_1,x_2}X^{p_1x_2+x_1}$
4: **return** $(f_0(X), f_1(X))$
5: **else if** m is odd **then**
6: $(f_0(X), \ldots, f_{m-2}(X)) = \Omega_{m-1}(M_0, \ldots, M_{m-2})$
7: $f_{m-1}(X) = \sum_{x_m=0}^{p_m-1}\sum_{x_{m-1}=0}^{p_{m-1}-1} M_{m-1}^{p_{m-1}-1-x_{m-1},x_m}$
8: $X^{P_0^{m-1}x_m+P_1^{m-2}x_{m-1}}$
9: **return** $(f_0(X), \ldots, f_{m-1}(X))$
10: **else if** m is even **then**
11: $(f_1(X), \ldots, f_{m-1}(X)) = \Omega_{m-1}(M_1, \ldots, M_{m-1})$
12: $f_0(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_1=0}^{p_1-1} M_0^{p_1-1-x_1,x_0}X^{P_1^mx_0+P_2^{m-1}x_1}$
13: **return** $(f_0(X), \ldots, f_{m-1}(X))$

B. Recovery Threshold

We first analyze the recovery threshold of the general coding scheme. From Alg. 1, we can see that $F_m(X)$ is a polynomial function of X, and thus we can write $F_m(X)$ as

$$F_m(X) = \sum_{x_0=0}^{p_0-1}\sum_{x_m=0}^{p_m-1}\sum_{s_1=0}^{2p_1-2}\cdots\sum_{s_{m-1}=0}^{2p_{m-1}-2} C_m(x_0, x_m, s_1, \ldots, s_{m-1})X^{R_m(x_0,x_m,s_1,\ldots,s_{m-1})},$$

where $C_m(x_0, x_m, s_1, \ldots, s_{m-1})$ and $R_m(x_0, x_m, s_1, \ldots, s_{m-1})$ denote the coefficient and exponent of each term, respectively.

As $F_m(X)$ is still a polynomial function of X, we can obtain the recovery threshold from the degree of $F_m(X)$, *i.e.*, by investigating the value of $R_m(x_0, x_m, s_1, \ldots, s_{m-1})$.

Lemma 1. *When m is odd,*

$$R_m(x_0, x_m, s_1, \ldots, s_{m-1}) = P_0^{m-1}x_m + P_1^{m-1}x_0 + \sum_{l=0}^{\frac{m}{2}-\frac{3}{2}} \left(P_{1+l}^{m-2-l}s_{m-1-l} + P_{2+l}^{m-2-l}s_{1+l}\right). \quad (6)$$

When m is even,

$$R_m(x_0, x_m, s_1, \ldots, s_{m-1}) = P_1^m x_0 + P_1^{m-1}x_m + P_2^{m-1}s_1 + \sum_{l=0}^{\frac{m}{2}-2} \left(P_{2+l}^{m-2-l}s_{m-1-l} + P_{3+l}^{m-2-l}s_{2+l}\right). \quad (7)$$

Proof. We prove this lemma with induction. The cases of $m = 2$ and $m = 3$ have been proved in Sec. III and Sec. IV.

When $m > 3$ and $m - 1$ is odd, m will be an even integer. From Line 11 of Alg. 1, we know that the exponent of X in each term in $\prod_{i=1}^{m-1} f_i(X)$ will be

$$P_1^{m-1}x_m + P_2^{m-1}x_1 + \sum_{l=0}^{\frac{m-1}{2}-\frac{3}{2}} \left(P_{2+l}^{m-2-l}s_{m-1-l} + P_{3+l}^{m-2-l}s_{2+l}\right) \quad (8)$$

Also, we have

$$f_0(X) \quad = \quad \sum_{x_0=0}^{p_0-1}\sum_{x_1=0}^{p_1-1} M_1^{p_1-1-x_1,x_0} X^{P_1^m x_0 + P_2^{m-1} x_1}.$$

Therefore, when we multiply $f_0(X)$ with $\prod_{i=1}^{m-1} f_i(X)$, $R_m(x_0, x_m, s_1, \ldots, s_{m-1})$ will be

$$R_m(x_0, x_m, s_1, \ldots, s_{m-1}) = P_1^m x_0 + P_2^{m-1} s_1 + P_1^{m-1} x_m + \sum_{l=0}^{\frac{m-1}{2}-\frac{3}{2}} \left(P_{2+l}^{m-2-l} s_{m-1-l} + P_{3+l}^{m-2-l} s_{2+l}\right), \quad (9)$$

which is equivalent as (7).

When $m-1$ is even and m is odd, the exponent in each term of $\prod_{i=0}^{m-2} M_i$ can be obtained from R_{m-1}, *i.e.*,

$$P_1^{m-1} x_0 + P_1^{m-2} x_{m-1} + P_2^{m-2} s_1 + \sum_{l=0}^{\frac{m-1}{2}-2} \left(P_{2+l}^{m-3-l} s_{m-2-l} + P_{3+l}^{m-3-l} s_{2+l}\right). \quad (10)$$

From Line 7 in Alg. 1, we know the exponent of each term in $f_m(X)$ is $P_0^{m-1} x_m + P_1^{m-2} x_{m-1}$, and thus the exponent of each term after the multiplication will become

$$\begin{aligned} & P_0^{m-1} x_m + P_1^{m-2} s_{m-1} + P_1^{m-1} x_0 + P_2^{m-2} s_1 + \sum_{l=0}^{\frac{m-1}{2}-2} \left(P_{2+l}^{m-3-l} s_{m-2-l} + P_{3+l}^{m-3-l} s_{2+l}\right) \\ = \; & P_0^{m-1} x_m + P_1^{m-1} x_0 + \sum_{l=0}^{\frac{m}{2}-\frac{3}{2}} \left(P_{1+l}^{m-2-l} s_{m-1-l} + P_{2+l}^{m-2-l} s_{1+l}\right). \end{aligned} \quad (11)$$

□

Lemma 2. *Exponents in $F_m(X)$ are consecutive.*

Proof. If $m = 2$, we can see that the range of all exponents in $f_1(X)$ is consecutive. Meanwhile, given a fixed x_0 in $f_0(X)$, the exponents in $\sum_{x_1=0}^{p_1-1} M_1^{x_0,x_1} X^{p_1 p_2 x_0 + x_1}$ are also consecutive. Hence, with a fixed x_0, all values of $R_2(x_0, x_2, s_1)$ are consecutive. From (7), we can also see that $R_2(x_0+1, 0, 0) = R_2(x_0, p_2-1, p_1-1)+1$. In other words, exponents with x_0 and x_0+1 are overlapped. Therefore, the exponents $R_2(x_2, x_2, s_1)$ with all values of x_0 are consecutive.

If all exponents in $F_{m-1}(X)$ are consecutive, we now prove that all exponents in $F_m(X)$ are also consecutive. If m is odd, from Line 7 in Alg. 1 we can get that if x_m is fixed, all corresponding exponents in $F_m(X)$ will be consecutive. We can also see that $R_m(0, x_m+1, 0, \ldots, 0) = R_m(p_0-1, x_m, p_1-1, \ldots, p_{m-1}-1)+1$. Similarly, if m is even, we can get that if x_0 is fixed, all corresponding exponents in $F_m(X)$ will be consecutive from Line 12 in Alg. 1. We can also get that $R_m(x_0, p_m-1, p_1-1, \ldots, p_{m-1}-1) = R_m(x_0+1, 0, \ldots, 0)+1$. By such overlappings, we know that the exponents in $F_m(X)$ are consecutive. □

With Lemma 1 showing the exponents of all terms in $F_m(X)$ and Lemma 2 showing that exponents in $F_m(X)$ are consecutive, we will now be able to obtain the recovery threshold by counting the number of all possible terms in this polynomial function.

Theorem 1. *In $F_m(X)$, $0 \le R_m(x_0, x_m, s_1, \ldots, s_{m-1}) \le P_0^m + P_1^{m-1} - 2$.*

Proof. It is easy to see that the minimal exponent in $F_m(X)$ is 0, when $x_0 = s_1 = \ldots = s_{m-1} = x_m$. We then only need to obtain the degree, *i.e.*, the maximal exponent in $F_m(X)$.

Since $x_i \le p_i - 1$, $i = 0, \ldots, m$, and $s_i \le 2p_i - 2$, $i = 1, \ldots, m-1$, the maximal exponent is

$$\begin{aligned} & P_0^{m-1}(p_m-1) + P_1^{m-1}(p_0-1) + \sum_{l=0}^{\frac{m}{2}-\frac{3}{2}} \left(P_{1+l}^{m-2-l}(2p_{m-1-l}-2) + P_{2+l}^{m-2-l}(2p_{1+l}-2)\right) \\ =& P_0^m - P_1^{m-1} + \sum_{l=0}^{\frac{m}{2}-\frac{3}{2}} \left(2P_{1+l}^{m-1-l} - 2P_{2+l}^{m-2-l}\right) \\ =& P_0^m - P_1^{m-1} + 2P_1^{m-1} - 2 \\ =& P_0^m + P_1^{m-1} - 2. \end{aligned}$$

Similarly, when m is even, the maximum exponent is

$$\begin{aligned} & P_1^m(p_0-1) + P_1^{m-1}(p_m-1) + P_2^{m-1}(2p_1-2) + \sum_{l=0}^{\frac{m}{2}-2} \left(P_{2+l}^{m-2-l}(2p_{m-1-l}-2) + P_{3+l}^{m-2-l}(2p_{2+l}-2)\right) \\ =\; & P_0^m + P_1^{m-1} - 2P_2^{m-1} + \sum_{l=0}^{\frac{m}{2}-2} \left(2P_{2+l}^{m-2-l} - 2P_{3+l}^{m-2-l}\right) \\ =\; & P_0^m + P_1^{m-1} - 2. \end{aligned}$$

□

Given the range of exponents in $F_m(X)$ above, we can get that there are at most $P_0^m + P_1^{m-1} - 1$ terms in $F_m(X)$. As shown in Lemma 2 that all possible exponents between 0 and $P_0^m + P_1^{m-1} - 1$ exist in the terms of $F_m(X)$, the recovery threshold is $P_0^m + P_1^{m-1} - 1$.

C. Correctness

We now demonstrate the correctness of our coding scheme. In other words, all submatrices in the overall result can be obtained after decoding.

Theorem 2. *For any $m \ge 2$, the $p_0 p_m$ submatrices in $\prod_{i=0}^{m-1} M_i$ can be found in $C_m(x_0, x_m, s_1, \ldots, s_{m-1})$ when $s_i = p_i - 1$, $i = 1, \ldots, m-1$.*

Proof. We also prove this theorem with induction. The cases of $m = 2$ and $m = 3$ have been proved in Sec. III and Sec. IV, respectively. We now prove that if this theorem is true with $m-1$ matrices, it is also true with m matrices.

If m is odd, we have $F_m(X) = F_{m-1}(X) f_{m-1}(X)$. From this equation, we know that the submatrices

in $\prod_{i=0}^{m-2} M_i$ can be found when $s_i = p_i - 1$, $i = 1, \ldots, m-2$. In other words, we define $\hat{F}_{m-1}(X) = \sum_{x_0=1}^{p_0} \sum_{x_{m-1}=1}^{p_{m-1}} C_{m-1}(x_0, x_{m-1}, p_1 - 1, \ldots, p_{m-2} - 1) X^{R_{m-1}}$, which is composed of the $p_0 p_{m-1}$ submatrices in $\prod_{i=0}^{m-2} M_i$. In particular, the values of x_0 and x_{m-1} in the coefficients determine the position of the corresponding submatrices. Therefore, the submatrices in $\prod_{i=0}^{m-1} M_i$ will also be generated from coefficients in

$$\begin{aligned}
&\hat{F}_{m-1}(X) f_{m-1}(X) \\
&= \sum_{x_0=0}^{p_0-1} \sum_{x_m=0}^{p_m-1} \sum_{s_{m-1}=0}^{2p_{m-1}-2} \sum_{x_{m-1}=\max\{0, s_{m-1}-p_{m-1}+1\}}^{\min\{p_{m-1}-1, s_{m-1}\}} \\
&C_{m-1}(x_0, x_{m-1}, p_1 - 1, \ldots, p_{m-2} - 1) \\
&M_m^{p_{m-1}-1-s_{m-1}+x_{m-1}, x_m} X^{R_{m-1} + P_0^{m-1} x_m + P_1^{m-2} x_{m-1}}.
\end{aligned}$$

In the equation above, if we fix $s_{m-1} = p_{m-1} - 1$, we have coefficients

$$\begin{aligned}
&\sum_{x_{m-1}=0}^{p_{m-1}-1} C_{m-1}(x_0, x_{m-1}, p_1 - 1, \ldots, p_{m-2} - 1) M_m^{x_{m-1}, x_m} \\
&\qquad = C_m(x_0, x_m, p_1 - 1, \ldots, p_m - 1). \quad (12)
\end{aligned}$$

From (12), we obtain all $p_0 p_m$ submatrices in $\prod_{i=0}^{m-1} M_i$, with $0 \le x_0 \le p_0 - 1$ and $0 \le x_m \le p_m - 1$.

On the other hand, when m is even, we have $F_m(X) = f_0(X) \cdot \left(\prod_{i=1}^{m-1} f_i(X) \right)$. The $p_1 p_{m-1}$ submatrices in $\prod_{i=1}^{m-1} M_i$ can then be found in $C_{m-1}(x_1, x_m, s_2, \ldots, s_{m-1})$, where $s_i = p_i - 1$, $i = 1, \ldots, m-1$. Hence, the submatrices in $\prod_{i=0}^{m-1} M_i$ can be generated from coefficients in

$$\begin{aligned}
&\sum_{x_1=0}^{p_1-1} M_0^{x_0, x_1} C_{m-1}(x_1, x_m, p_2 - 1, \ldots, p_{m-1} - 1) \\
&\qquad = C_m(x_0, x_m, p_1 - 1, \ldots, p_m - 1).
\end{aligned}$$

□

Moreover, from the proof of Lemma 1, we can show that the exponents of desired coefficients do not overlap with those of noise coefficients in Theorem 3, which completes the proof of the correctness of our coding scheme with Theorem 2.

Theorem 3. *The exponents of the $p_0 p_m$ desired coefficients are unique in $F_m(X)$.*

Proof. As shown in Fig. 2 and Fig. 3, two coefficients may have the same exponent with different parameters in $R_m(x_0, x_m, s_1, \ldots, s_{m-1})$. We prove this theorem by proving that changing any parameter in $R_m(x_0, x_m, s_1, \ldots, s_{m-1})$ will not lead to an overlap with an exponent of a desired coefficient.

When $m = 2$, we can see from (1) that the exponent of $(x_0, x_2 + 1, s_1)$ equals that with $(x_0, x_2, s_1 + p_1)$ if $x_2 < p_2 - 1$. When $x_2 = p_2 - 1$, the exponent with $(x_0, p_2 - 1, s_1)$ also equals that with $(x_0 + 1, 0, s_1 + p_1)$. If $s_1 = p_1 - 1$, $s_1 + p_1 > 2p_1 - 2$ is invalid. In addition, $s_1 + p_1$ is impossible to equal $p_1 - 1$. Therefore, the exponent of a desired coefficient will not overlap with that of any other coefficient.

Similarly, we can also see from (5) that the exponent with $(x_3, x_0 + 1, s_2, s_1)$ equals that with $(x_3, x_0, s_2 + p_2, s_1)$ (when $x_0 < p_0 - 1$), and the exponent with $(x_3, x_0, s_2 + 1, s_1)$ equals that with $(x_3, x_0, s_2, s_1 + p_1)$ (when $s_2 < 2p_2 - 2$). Still, none of the overlapped exponents above can belong to a desired coefficient. The overlapped exponent can also be found for the case of $x_0 = p_0 - 1$ or $s_2 = 2p_2 - 2$. In the rest of this proof, we will only consider the cases where $x_i < p_i - 1$ or $s_i < 2p_i - 2$, the proof for the case where $x_i = p_i - 1$ or $s_i = 2p_2 - 2$ can be obtained similarly.

In order to prove the case of a general m, we first reorder the parameters in $R_m(x_0, x_m, s_1, \ldots, s_{m-1})$ by the degrees of their coefficients. In (6), the parameters can be ordered as $(x_m, x_0, s_{m-1}, s_1, \ldots, s_{\frac{m+1}{2}}, s_{\frac{m-1}{2}})$. Similarly, the parameters can be ordered as $(x_0, x_m, s_1, s_{m-1}, \ldots, s_{\frac{m}{2}-1}, s_{\frac{m}{2}})$ in (7). Consider an adjacent pair of parameters b_i, which is either x_i or s_i ($i \neq m$ when m is odd, and $i \neq 0$ when m is even), and s_j. Examples of such pairs include x_0 and s_{m-1} when m is odd, and s_1 and s_{m-1} when m is even. Given any such adjacent pair, this theorem can be proved if we can prove that the exponent with $(\ldots, b_i, s_j, \ldots,)$ equals that with $(\ldots, b_i, s_j + p_j, \ldots)$. This theorem is already proved for $m = 2$ or $m = 3$.

We finally prove this theorem with $m > 3$ using induction. If $m - 1$ is odd, comparing (8) and (9), we can see that the coefficients $(x_1, s_{m-1}, s_2, \ldots)$ in (8) are inherited as $(s_1, s_{m-1}, s_2, \ldots)$ in (9). Hence, if this theorem is true with $m - 1$, then it is true with m when b_i is in $(s_1, s_{m-1}, s_2, \ldots)$. So now we only need to consider the case of $b_i = x_m$, and then $b_j = s_1$. We can see that the exponent of $(x_0, x_m + 1, s_1, \ldots)$ equals that of $(x_0, x_m, s_1 + p_1, \ldots)$.

When $m - 1$ is even, we can also see that the weights of $(x_{m-1}, s_1, s_{m-2}, s_2, \ldots)$ in (10) are inherited by those of $(s_{m-1}, s_1, s_{m-2}, s_2, \ldots)$ in (11). Hence, we can also prove that the exponents between $(x_m, x_0 + 1, s_{m-1}, \ldots)$ and $(x_m, x_0, s_{m-1} + p_{m-1}, \ldots)$ are the same, finishing the proof. □

VI. Evaluation

In this section, we present our empirical results of running the coded matrix multiplication in a cluster of virtual machines hosted on Microsoft Azure. All coded tasks run on virtual machines of type B1. The job is controlled by another virtual machine of type B4 as a master, which also decodes the results of tasks as decoding requires more memory than each task. We first evaluate the performance of the matrix chain multiplication, and then solve a linear regression problem with the matrix chain multiplication.

A. Matrix Chain Multiplication

We implement our coding scheme (chain) for the matrix chain multiplication with OpenMPI. The m coded matrices in $\Omega_m(M_0, \ldots, M_{m-1})$ are initially stored on each worker. Each worker multiplies such m coded matrices and uploads

the result to the master. The master keeps polling if there is any new result sent from a worker, terminates all remaining tasks once the number of received results reaches the recovery threshold, and then decodes the results. In our experiments, we decode the results with Gaussian elimination. Note that we only need to obtain the $p_0 p_m$ desired coefficients in $F_m(X)$, and thus the decoding will be stopped once we get such desired coefficients in order to save time.

As a comparison, we implement another scheme EP (partition) which completes coded chain multiplication in multiple rounds, each of which is encoded with an entangled polynomial code. To make a fair comparison, we also first store each m matrices encoded with entangled polynomial codes on each worker. In the first round, each worker multiplies the two coded matrices in $\Omega_2(M_0, M_1)$, and the master will obtain $P_1 = M_0 M_1$. Then the master will only encode P_1 as in $\Omega_2(P_1, M_2)$, as M_2 has already been encoded, and sends coded matrices to each worker. In this round, $P_2 = P_1 M_2$ will be calculated and the master will also encode P_2, and so on until all m input matrices have been multiplied at the end of the $(m-1)$-th round.

We run jobs with the two schemes above, which multiply m random matrices of the same size with $m = 3$, 4, and 5, respectively. In these jobs, the sizes of the m matrices are 2000×2000 and 4000×4000. Each matrix is split both vertically and horizontally into 2 partitions, *i.e.*, $p_0 = \cdots = p_m = 2$. Therefore, the recovery thresholds for the jobs in chain are 19 ($m = 3$), 39 ($m = 4$), and 79 ($m = 5$). The number of workers is then chosen as the sum of the corresponding recovery threshold and 5 additional workers, such that at most 5 stragglers can always be tolerated. When running the job with entangled polynomial codes in multiple rounds, the number of workers in each round will be chosen such that the same number of stragglers can be tolerated.

Although EP (partition) maintains the same partitions as chain, the entangled polynomial code has a much lower recovery threshold and thus require much fewer workers if the same number of stragglers need to be tolerated. Hence, we run the same jobs with one more scheme EP (worker), which is also based on the entangled polynomial code, by increasing the number of partitions of input matrices in each round so that the same number of workers will be required as chain. The numbers of partitions in the two input matrices in each round will be 2×4 and 4×2 ($m = 3$), 3×4 and 4×3 ($m = 4$), and 3×8 and 8×3 ($m = 5$). If the rows or columns of a matrix cannot be equally divisible by the number of partitions, we will add additional zero rows or columns at the end of the matrix.

In Fig. 4, we present the job completion time of coded chain multiplication in the two jobs with the three schemes above, *i.e.*, chain, EP (partition), and EP (worker). We run each job 60 times, and show the average of its job completion time in Fig. 4. We can see that our coding scheme can significantly save the overall job completion time by at most 90.3%. Compared to EP (partition), although more tasks are allowed with our coding scheme, making the parallelism of

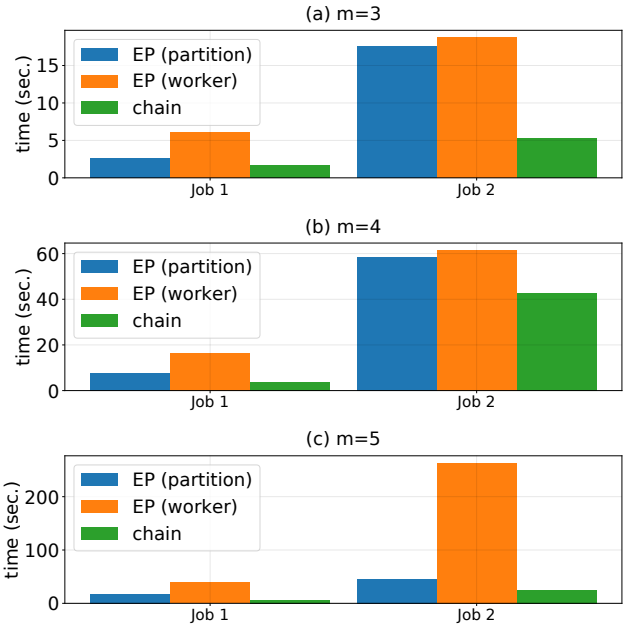

Fig. 4. Comparison of the job completion time of the matrix chain multiplications.

the job increase, the most saving of time comes from the communication overhead, as with entangled polynomial codes the intermediate results should be uploaded to the master and then be encoded for the next round. Moreover, although EP (worker) enjoys the same level of parallelism as chain, its high communication overhead (as illustrated in Fig. 5) actually becomes its bottleneck, and we can see that its job completion time is the worst.

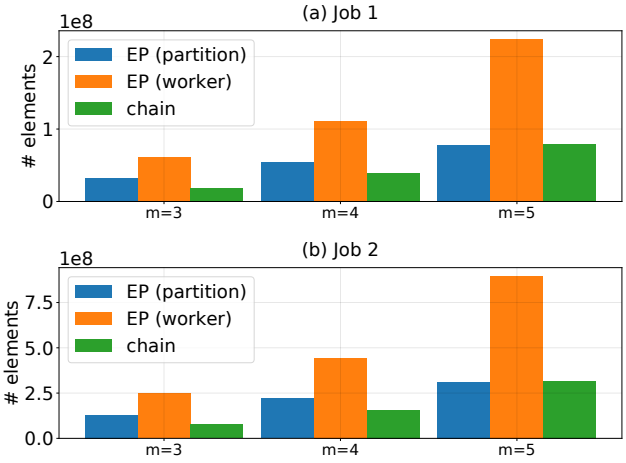

Fig. 5. Comparison of the communication overhead of the matrix chain multiplications.

We also compare the communication overhead in Fig. 5. We measure the total numbers of elements in the matrices transferred over the network in the jobs running with the three

schemes. We can see that compared to EP (partition), the communication overhead is more significantly saved by chain when the value of m is smaller. In fact, our calculation shows that when $m = 6$ the communication overhead of chain will be higher than EP. However, such traffic will be sent in parallel in one round only, while the traffic in EP (partition) need to be sent sequentially in multiple rounds.

Moreover, we note that EP (partition) maintains the same partitions as chain and thus requires much fewer workers than chain. As for EP (worker), it maintains the same number of workers by increasing the number of partitions, and we can observe in Fig. 5 that the overall traffic is significantly higher than the other two schemes.

From our experiments and analysis, we can see that if m is very large, the network overhead of the coded matrix chain multiplication can also be large, as the recovery threshold increases exponentially. In such cases, we may limit the number of partitions in some small input matrix, or split the whole matrix chain multiplication into multiple chain multiplication with fewer input matrices. However, in this paper we focus on the general mechanism for the coded matrix chain multiplication, and the optimal choices of partitioning is out of the scope of this paper. Therefore, we leave such discussions in our future work.

B. Linear Regression

We now use the coded matrix chain multiplication to solve a linear regression problem in a distributed manner. The problem is modeled as $\min_x f(x) \triangleq \min_x \frac{1}{2}||Ax-y||^2$, where $y \in \mathbb{R}^q$ is the label vector, $A \in \mathbb{R}^{q\times r}$ is the matrix of the dataset, and $x \in \mathbb{R}^r$ is the unknown weight vector to be trained. We solve the linear regression problem with gradient descent. Initially, the weight vector is set as $x^{(0)}$. We then update it iteratively as $x^{(t+1)} = x^{(t)} - \gamma\nabla f(x^{(t)}) = x^{(t)} - \gamma A^T(Ax^{(t)} - y)$, $t \geq 0$. We can then observe that each step can be completed by two matrix multiplications, *i.e.*, $g^{(t)} \triangleq Ax^{(t)}$, and $A^T(g^{(t)} - y)$. Hence, we first build a distributed straggler-free solver for the linear regression problem. To tolerate potential stragglers, we use entangled polynomial codes to encode A and $x^{(t)}$ in the first matrix multiplication, and A^T and $g^{(t)} - y$ in the second matrix multiplication, and then proceed to the next step. As A and A^T do not change in each step, we can place their coded matrices on each worker before the job starts, and hence only $x^{(t)}$ and $g^{(t)} - y$ need to be encoded and sent to all workers.

The algorithm above was also used by Lee *et al.* [3] and Yang *et al.* [32], requiring two rounds of matrix multiplications in each step. However, we observe that $x^{(t+1)}$ can be written as $x^{(t+1)} = x^{(t)} - \gamma A^T Ax^{(t)} + \gamma A^T y$. Since A and y are constant, we only need to compute $A^T y$ once. In each step, we then only need to compute a matrix chain multiplication $A^T Ax^{(t)}$. Hence, we implement another distributed solver based on the coded matrix chain multiplication with $m = 3$. We can see that compared to existing solvers based on the distributed matrix multiplication, the number of matrix multiplications in each step is saved from two to one. Still, A and A^T are encoded and placed on each worker before the job starts, and only $x^{(t)}$ need to be encoded and sent to all worker per step.

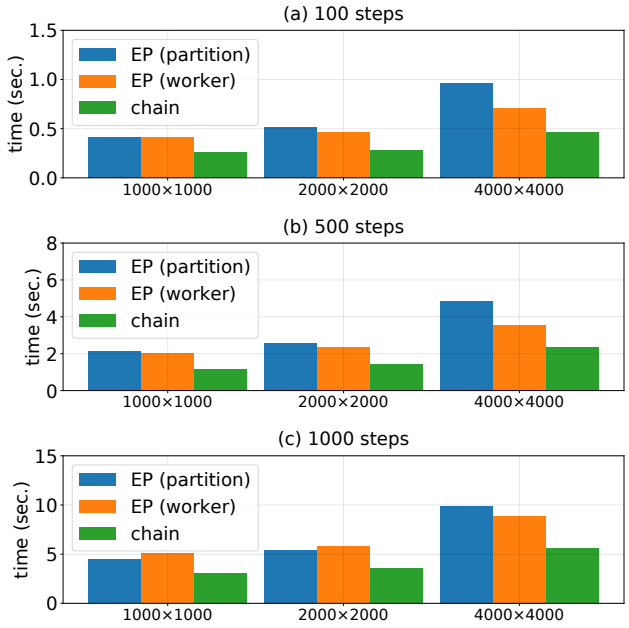

Fig. 6. Comparison of the job completion time of the linear regression.

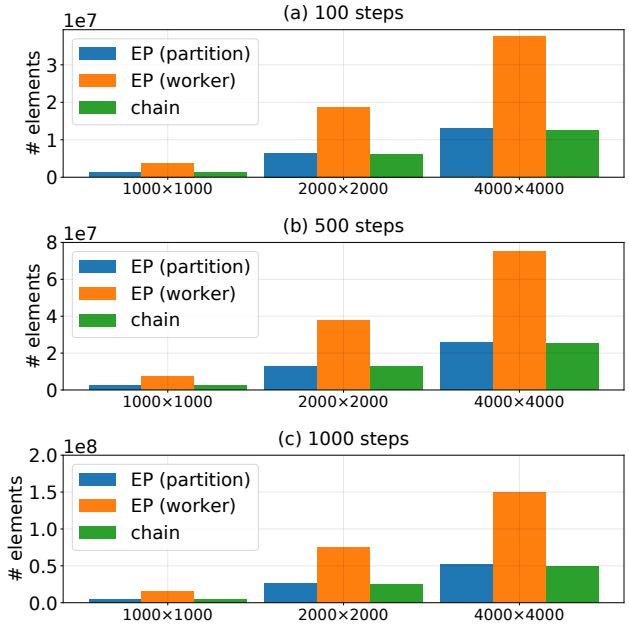

Fig. 7. Comparison of the communication overhead of the linear regression.

In our experiment, we run the same jobs using the two solvers above, *i.e.*, EP and chain. We use coded matrix chain multiplication in chain. In chain, A and A^T are both partitioned into 2×2 submatrices. As $x^{(t)}$ is a vector, we just split it into 2 partitions horizontally. The same as in

Sec. VI-A, we split the matrices in two ways in EP. EP (partition) partitions the input matrices in the same way as in chain, and EP (worker) increases the input matrices so that the recovery threshold equals that in chain, *i.e.*, A and A^T are partitioned into 4×2 submatrices and $x^{(t)}$ is still split into 2 partitions. We repeat each job 20 times, and obtain their average time of completion. We also choose the number of steps in each job to be 100, 500, and 1000, respectively. The sizes of the dataset matrix A are randomly generated in three different sizes, 1000×1000, 2000×2000, and 4000×4000, as well as the label vector y with the corresponding sizes.

Fig. 6 illustrates the completion time of the jobs running with the three schemes above. Comparing to EP (partition), chain is faster by up to 51.7%. EP (worker) is also a bit faster than EP (partition), by up to 27.29%, due to its higher parallelism. Although EP (worker) enjoys the same recovery threshold as chain, chain is still faster since each step only requires one chain multiplication, leading to a higher level of parallelism since the two matrix multiplications in each step in EP (worker) can only be done sequentially.

Besides the job completion time, we also compare the corresponding communication overhead of the three schemes in Fig. 7. The same as in Fig. 5, we measure the total numbers of elements in the matrices/vectors transferred over the network in the jobs running with the three schemes. We can see that the traffic sent over the network in chain is actually very similar to that in EP (partition), only 3.85% percent lower, mainly because they partition the input data the same way. Hence, it is the saving of matrix multiplications that lead to the saving of job completion time. It is also interesting to observe that EP (worker) incurs 1.88 time more communication overhead than EP (partition). In other words, the saving of complexity of the tasks of EP (worker) compensates for the additional time of communication. On the other hand, as the sizes of coded matrices/vectors are the same between EP (worker) and chain, we can infer that the higher communication overhead, as well as the two multiplications in each step, make EP (worker) up to 71.0% slower than chain.

VII. Conclusion

Coded computing for the distributed matrix multiplication have been demonstrated to efficiently tolerate stragglers. However, existing coding schemes proposed so far have only considered the multiplication of two matrices, while in practice learning-based algorithms commonly require the multiplication of multiple large matrices. As the existing coded matrix multiplication can only multiply two matrices each time, with which the chain matrix multiplication needs to be completed in multiple rounds, we propose a coding scheme for the matrix chain multiplication with a general number of matrices multiplied, which allows to complete the chain multiplication in one single round. Our experiments on Microsoft Azure demonstrate that it significantly saves the time of the distributed matrix chain multiplication and distributed linear regression, compared to multiplying two matrices each time.

Acknowledgment

This paper is based on work supported by the National Science Foundation under Grant No. CCF-2101388.

References

[1] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," in *USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2017.

[2] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient Coding: Avoiding Stragglers in Distributed Learning," in *International Conference on Machine Learning (ICML)*, 2017, pp. 3368–3376.

[3] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding Up Distributed Machine Learning Using Codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[4] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.

[5] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[6] S. Dutta, V. Cadambe, and P. Grover, "Short-Dot: Computing Large Linear Transforms Distributedly Using Coded Short Dot Products," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 2100–2108.

[7] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Straggler Mitigation in Distributed Matrix Multiplication: Fundamental Limits and Optimal Coding," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 2022–2026.

[8] A. Reisizadeh, S. Prakash, R. Pedarsani, and S. Avestimehr, "Coded Computation over Heterogeneous Clusters," in *IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2408–2412.

[9] S. Kiani, N. Ferdinand, and S. C. Draper, "Exploitation of Stragglers in Coded Computation," in *IEEE International Symposium on Information Theory (ISIT)*, 2018.

[10] T. Baharav, K. Lee, O. Ocal, and K. Ramchandran, "Straggler-proofing Massive-scale Distributed Matrix Multiplication with d-dimensional Product Codes," in *IEEE International Symposium on Information Theory (ISIT)*, 2018, pp. 1993–1997.

[11] S. Wang, J. Liu, and N. Shroff, "Coded Sparse Matrix Multiplication," in *International Conference on Machine Learning (ICML)*, 2018.

[12] V. Gupta, S. Wang, T. Courtade, and K. Ramchandran, "OverSketch: Approximate Matrix Multiplication for the Cloud," in *IEEE International Conference on Big Data*, 2018.

[13] H. Park, K. Lee, J.-Y. Sohn, C. Suh, and J. Moon, "Hierarchical Coding for Distributed Computing," in *IEEE International Symposium on Information Theory (ISIT)*, 2018.

[14] Q. Yu, M. A. Maddah-Ali, and A. S. Avestimehr, "Polynomial Codes: an Optimal Design for High-Dimensional Coded Matrix Multiplication," *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[15] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the Optimal Recovery Threshold of Coded Matrix Multiplication," *IEEE Transactions on Information Theory*, vol. 66, no. 1, pp. 278–301, 2019.

[16] N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding Structure with Randomness : Probabilistic Algorithms for Matrix Decompositions," *SIAM Review*, 2011.

[17] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted Boltzmann Machines for Collaborative Filtering," in *International Conference on Machine Learning (ICML)*, 2008, pp. 791–798.

[18] A. Tulloch, "Fast Randomized SVD," 2014. [Online]. Available: https://research.fb.com/fast-randomized-svd/

[19] S. S. Godbole, "On Efficient Computation of Matrix Chain Products," *IEEE Transactions on Computers*, vol. 22, no. 9, pp. 864–866, 1973.

[20] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[21] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[22] "Distributed Training with Keras," 2019. [Online]. Available: https://www.tensorflow.org/beta/tutorials/distribute/keras

[23] A. Sergeev and M. Del Balso, "Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow," 2017. [Online]. Available: https://eng.uber.com/horovod/

[24] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones," in *Advances in Neural Information Processing Systems (NIPS)*, 2013, pp. 185–198.

[25] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[26] N. B. Shah, K. Lee, and K. Ramchandran, "When Do Redundant Requests Reduce Latency?" *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 715–722, 2016.

[27] Z. Qiu and J. F. Pérez, "Evaluating Replication for Parallel Jobs: An Efficient Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2288–2302, 2016.

[28] D. Wang, G. Joshi, and G. Wornell, "Efficient Task Replication for Fast Response Times in Parallel Computation," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 599–600, 2014.

[29] K. Lee, R. Pedarsani, and K. Ramchandran, "On Scheduling Redundant Requests with Cancellation Overheads," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1279–1290, 2017.

[30] A. Mallick, M. Chaudhari, and G. Joshi, "Rateless Codes for Near-Perfect Load Balancing in Distributed Matrix-Vector Multiplication," Tech. Rep., 2018. [Online]. Available: http://arxiv.org/abs/1804.10331

[31] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional Coded Matrix Multiplication," in *IEEE International Symposium on Information Theory (ISIT)*, 2017.

[32] Y. Yang, M. Interlandi, P. Grover, S. Kar, S. Amizadeh, and M. Weimer, "Coded Elastic Computing," in *IEEE International Symposium on Information Theory*, 2019.